

# Funzioni e moduli

Andrea Passerini  
passerini@disi.unitn.it

Informatica

## Descrizione

- Una funzione è una sequenza di istruzioni cui viene dato un nome
- Questo permette di riutilizzare tale sequenza di istruzioni in più parti del proprio programma (o in altri programmi) senza doverla scrivere ogni volta
- Inoltre, è possibile modificare la funzione e l'effetto si ripercuoterà dovunque venga usata, semplificando la manutenzione e lo sviluppo del codice.
- Una funzione può prendere in ingresso uno o più *argomenti*, o *parametri*
- Una funzione può restituire in uscita un risultato

## Uso

- Problemi complessi devono essere decomposti via via in problemi sempre più semplici per poter essere risolti
- Tale concetto è alla base della programmazione `strutturata`
- Le funzioni permettono di implementare agevolmente tale decomposizione, occupandosi ciascuna di sottoproblemi specifici
- Ovviamente una funzione può chiamare altre funzioni al suo interno per risolvere dei sottoproblemi
- Inoltre, una funzione sufficientemente generica può essere usata in vari contesti diversi

## Descrizione

- Una funzione è un tipo particolare di statement composto:

```
def <name> (arg1, arg2, ..., argN) :  
    <statements>
```

- Come tutti gli statements composti, consta di un'intestazione terminata da ":" e di un blocco indentato di statement annidati, e finisce quando finisce l'indentazione

**def** definisce lo statement come la creazione di una funzione

**name** è il nome della funzione, con cui dovrà essere chiamata per usarla

**arg1, arg2, ..., argN** sono gli argomenti della funzione (da zero ad un numero arbitrario)

**statements** sono il corpo della funzione, che ne esegue le operazioni

## Descrizione

- Se la funzione ha degli ingressi, il suo corpo tipicamente eseguirà delle operazioni che li coinvolgono

```
def print_max(l):  
    max = l[0]  
    for e in l[1:]:  
        if e > max:  
            max = e  
    print max
```

- Una funzione può restituire un oggetto in uscita tramite lo **statement** `return`

```
def max(l):  
    max = l[0]  
    for e in l[1:]:  
        if e > max:  
            max = e  
    return max
```

## Descrizione

- Lo statement `return` non deve essere necessariamente l'ultimo nel corpo della funzione, né essercene uno solo
- Nell'esecuzione, la funzione terminerà la prima volta che esegue uno statement `return`, restituendo l'oggetto relativo.

```
def ha_negativo(l):  
    for e in l:  
        if e < 0:  
            return True  
    return False
```

- Le funzioni che non hanno `return` (o che terminano senza averne incontrato nessuno) restituiscono `None` di default

## Descrizione

- Per le funzioni che restituiscono un risultato, questi tipicamente sarà usato in un'espressione, o assegnato direttamente ad una variabile

```
l = [2, 3, 5, 1, 2, 3]
```

```
m = max(l)
```

```
n = m - max([3, 4, 5])
```

```
if ha_negativo(l):  
    print "trovato negativo"
```

## Attenzione

- E' un errore molto comune assegnare ad una variabile il risultato di una funzione che restituisce `None`
- L'errore si commette facilmente con i metodi di modifica di lista o dizionario, che modificano l'oggetto ma restituiscono `None`

```
>>> L = [0,1,2,3]
>>> L = L.extend([4,5])
>>> L
>>>      # L contiene None!
```

## Restituire più oggetti

- E' possibile restituire un numero arbitrario di oggetti, mettendoli in una tupla

```
def max_min(l):  
    max = min = l[0]  
    for e in l[1:]:  
        if e > max:  
            max = e  
        elif e < min:  
            min = e  
    return max,min
```

## Restituire più oggetti

- Il risultato può come sempre essere assegnato ad una tupla o direttamente alle sue variabili:

```
l = [2, 3, 1, 5, 6]
```

```
t = max_min(l)  
max, min = max_min(l)
```

```
print t  
print max, min
```

**produce**

```
(6, 1)  
6 1
```

## Descrizione

- Gli argomenti di una funzione sono nomi di variabili che la funzione utilizzerà al suo interno
- Nel chiamare una funzione, gli argomenti vengono passati specificando oggetti o variabili all'interno delle parentesi dopo il nome della funzione, nello stesso ordine con cui appaiono nello statement della funzione
- Il risultato è che l'oggetto o la variabile vengono assegnati alle variabili della funzione nelle posizioni corrispondenti

## Esempio

```
def stampa(a1, a2, a3):  
    print a1, a2, a3
```

```
l = [0,2]
```

```
s = "av"
```

```
stampa(l, s, (1,2,3))
```

produce

```
[0, 2] av (1, 2, 3)
```

## Polimorfismo

- La funzione `stampa` può prendere in ingresso oggetti di qualunque tipo, finché possono essere stampati tramite `print`
- Tale caratteristica è chiamata *polimorfismo*, ed è la stessa che permette di sommare due stringhe o due numeri con l'operatore `+`
- Polimorfismo significa che il risultato di un'operazione (o di una funzione) dipende dal tipo degli operandi (o degli argomenti)
- Questo permette di scrivere un'unica funzione che opera su oggetti di vari tipi diversi, ammesso che le operazioni eseguite al suo interno siano compatibili con i tipi degli argomenti

## Esempio

- Somma di lista di interi, o concatenazione di stringhe

```
def somma(l):  
    res = l[0]  
    for e in l[1:]:  
        res += e  
    return res  
  
i = somma([1,2,3,4])  
s = somma(["a", "b", "c", "d"])  
  
print i,s  
  
produce  
  
10 abcd
```

## argomenti mutabili

- Se un argomento passato ad una funzione è di tipo mutabile, le modifiche ad esso fatte all'interno della funzione si ripercuoteranno all'oggetto "esterno" (la funzione manipola un riferimento a tale oggetto)

```
def elimina_valore(d, val):  
    for (k,v) in d.items():  
        if v == val:  
            del d[k]
```

```
d = {"a" : 1, "b": 2, "c" : 2, "d" : 3}  
elimina_valore(d,2)  
print d
```

produce

```
{'a': 1, 'd': 3}
```

## Scope

- Tutte le variabili definite (assegnando loro un valore) all'interno di una funzione sono *locali* ad essa
- Una variabile locale è visibile solo all'interno della funzione (il suo spazio di visibilità o *scope*)
- Una variabile non può essere usata al di fuori del suo spazio di visibilità

```
def somma(l):  
    res = l[0] # variabile locale  
    for e in l[1:]:  
        res += e  
    return res
```

```
r = somma([1,2,4])  
print res # errore! variabile fuori scope
```

## Scope locale e globale

- Lo scope è un concetto fondamentale della programmazione strutturata
- La visibilità locale delle funzioni permette un elevato grado di indipendenza rispetto ai programmi in cui saranno chiamate (non ci si deve preoccupare se i nomi di variabili in esse definiti sono già usati altrove)
- Esiste uno *scope globale* che è esterno alle funzioni, e copre tutto il file in cui le funzioni si trovano (o vengono chiamate)

## Scope locale e globale

- Le variabili definite al di fuori delle funzioni, sono visibili anche al loro interno

```
def globali():  
    print x,y
```

```
x,y = 1,2  
globali()
```

**produce**

```
1 2
```

## Variabili globali

- E' buona programmazione non usare MAI variabili globali all'interno delle funzioni
- Una funzione dovrebbe comunicare con l'esterno SOLO tramite argomenti in ingresso e risultati in uscita
- Usare variabili globali rende la funzione difficilmente comprensibile, non essendo caratterizzata totalmente da ingressi ed uscite
- L'uso di variabili globali contrasta con la programmazione strutturata ed è spesso fonte di errori

## Variabili globali vs variabili locali

- Se una variabile locale ha lo stesso nome di una variabile globale esistente, la prima *nasconde* la seconda, che è come se non esistesse per la funzione

```
def somma(l):  
    res = l[0]  
    for e in l[1:]:  
        res += e  
    return res
```

```
res = 0  
r = somma([1,2,4])  
print r,res
```

produce

7 0

## Argomenti

- Gli argomenti di una funzione vengono assegnati alle variabili *locali* nelle posizioni corrispondenti
- Quindi la variabile locale e quella passata come argomento sono *diversi* riferimenti allo *stesso* oggetto (anche se hanno lo stesso nome!)
- Riassegnare alla variabile locale un oggetto diverso non modifica in alcun modo la variabile esterna

## Esempio

```
def cambia(l):  
    l = "stringa"  
    print l
```

```
l = [1,2,3]  
cambia(l)  
print l
```

**produce**

```
stringa  
[1, 2, 3]
```

## `help` e funzioni

- Abbiamo visto che la funzione `help` fornisce informazioni utili su funzioni e metodi
- `help` recupera tali informazioni dall'intestazione della funzione, e da una stringa di testo scritta all'inizio (come prima istruzione della funzione) se presente
- Documentare il codice con appropriati commenti è sempre molto utile per capirlo successivamente o farlo capire ad altri, e la stringa di documentazione è un aspetto che rende le funzioni Python facilmente auto-esplicative.

## Esempio

```
def somma(l):  
    "somma il contenuto di una lista"  
    res = l[0]  
    for e in l[1:]:  
        res += e  
    return res
```

```
print help(somma)
```

## produce

```
Help on function somma in module __main__:
```

```
somma(l)
```

```
    somma il contenuto di una lista
```

## Descrizione

- Nella sua forma più semplice, un modulo è un file di testo contenente codice Python, e salvato con estensione “.py”
- Un modulo contiene tipicamente funzioni o altri oggetti di utilità che possano essere usati dai programmi
- Una grandissima quantità di moduli sono stati realizzati per gli usi più svariati (vedremo)
- Esiste un set di moduli standard che è disponibile di default nella maggior parte delle installazioni di Python (e.g. `pickle`, `re`)

## Funzionamento

- Per poter essere utilizzato in un programma, un modulo deve essere *importato*
- Abbiamo visto come importare moduli con lo statement `import`

```
>>> import pickle
>> f = open("tmp", "w")
>>> pickle.dump(s, f)
```

## Ricerca

- Il file contenente il modulo `pickle` è stato individuato da Python perché la directory che lo contiene è tra quelle che Python ricerca durante un `pickle` (è un modulo di standard)
- Le librerie di moduli che installerete se necessario aggiorneranno questo spazio di ricerca, ed i moduli potranno essere caricati allo stesso modo
- Moduli creati da voi potranno essere facilmente importati allo stesso modo se nella stessa directory del file che li importa (esistono altri modi che non vedremo)

## Uso

- Una volta importato, un modulo può essere usato, ad esempio chiamando le sue funzioni
- Per poter chiamare la funzione di un modulo, deve essere preceduta dal nome del modulo seguito da punto:

```
>>> pickle.dump(s, f)
```

- Questo perché `dump` è un metodo dell'oggetto `pickle`
- Allo stesso modo, in `l.append()` `append` è un metodo di un oggetto di tipo lista

## Uso

- In un programma di dimensioni ragionevoli, esiste un file principale da cui parte l'esecuzione
- ed una serie di moduli che forniscono le funzionalità necessarie (possibilmente utilizzabili anche da altri programmi)
- tali moduli tipicamente contengono funzioni e/o altri oggetti, ma non codice che produca risultati se eseguiti direttamente, poiché sono fatti per essere usati da altri programmi
- sia il file principale sia i moduli possono importare ed utilizzare anche moduli esterni, tipo il set standard o altri

## Esempio

- File `utility.py` (da usare come modulo):

```
def somma(l):  
    res = l[0]  
    for e in l[1:]:  
        res += e  
    return res  
  
def max_min(l):  
    max = min = l[0]  
    for e in l[1:]:  
        if e > max:  
            max = e  
        elif e < min:  
            min = e  
    return max,min
```

## Esempio

- File `run.py` (da usare come file principale):

```
import utility

l = [2,4,5,6,1,3]
s = utility.somma(l)
max,min = utility.max_min(l)
print s,max,min
```

- Risultato dell'esecuzione di `run.py`:

```
21 6 1
```

## Statement `from`

- A volte risulta pesante dover sempre precedere una funzione con il nome del modulo che la contiene
- Per poter chiamare una funzione direttamente, basta importarla con lo statement `from`:

```
>>> from <nomemodulo> import <funzione>
```

- Se `<funzione>` viene sostituito con `*`, tutte le funzioni del modulo possono essere chiamate direttamente

## Esempio

```
from utility import somma

l = [2,4,5,6,1,3]
s = somma(l)
max,min = utility.max_min(l)
print s,max,min
```

oppure

```
from utility import *

l = [2,4,5,6,1,3]
s = somma(l)
max,min = max_min(l)
print s,max,min
```